Polly: A Language-Based Approach for Custom Change Detection of Web Service Data

Elyas Ben Hadj Yahia, Jean-Rémy Falleri, Laurent Réveillère

Polly: A Language-Based Approach for Custom Change Detection of Web Service Data

Elyas Ben Hadj Yahia[1,3], Jean-Rémy Falleri[2], and Laurent Réveillère[2]

[1] Univ. Bordeaux - LaBRI - UMR CNRS 5800, Talence, France
[2] Univ. Bordeaux - ENSEIRB-MATMECA Bordeaux INP - LaBRI - UMR CNRS 5800, Talence, France
[3] CProDirect, F-33700, Mérignac, France

Abstract. An ever-growing number of web service providers expose data that is continuously changing. Use cases arise where being notified about changes made to the data is essential to the client, for instance to know when a user has a new follower on Twitter. Monitoring changes on web services data consists in polling services for the required data, detecting any changes in the targeted data subset, and notifying the user only about the relevant changes. However, each step of this process can be relatively complex, leading to a tedious and challenging implementation for developers. In this paper we introduce Polly, a domain-specific language for describing change detection strategies in JSON data fetched from REST web APIs. By leveraging the domain knowledge of the user, our domain-specific language offers declarative, concise yet highly-expressive constructs for specifying change detection strategies. We validate our approach using several user-driven scenarios provided by our industrial partner and show that it outperforms the state-of-the-art solutions.

Keywords: DSL, Change Detection, API, REST, JSON.

1 Introduction

Integration platforms such as IFTTT[4] and Zapier[5] have recently emerged with the aim of orchestrating interactions between a multitude of web services such as Facebook and Twitter [11, 13]. They enable end users to describe which actions to trigger when a custom event occurs [16]. For instance, one may want to automatically tweet a message when a specific subway line becomes unavailable. However, most of existing web services do not provide a way to specify custom event notifications. To overcome this limitation, platform owners have developed their own notification system by performing a recurrent polling of monitored services. For each service, the current state is periodically fetched and compared against the previous one to identify specific values that vary over time. When a change is detected, the corresponding event is raised. Because specific code

[4] `https://ifttt.com`
[5] `https://zapier.com`

needs to be developed for each event of a service, the set of supported services and events is limited and does not necessarily meet user expectations.

Each step of the monitoring process can be relatively complex. As an example, consider the use of the Facebook service to detect new photos with a given tagged user in a given album. To implement this scenario, one needs first to periodically poll several Facebook API endpoints (the one for the photos and the one for the tags) and navigate through the paginated responses. The resulting aggregated state is then compared against the previous one. However, this comparison requires focusing only on new photos (identified by their unique IDs) while ignoring other irrelevant changes such as the last update time. Even such a simple use case underlines the complexities of this process, which are declined in two different challenges: state computation and change detection.

Although the computation of a state sometimes requires fetching a unique resource from a single API endpoint, it is often necessary to implement more complicated policies. For instance, the construction of a state may require navigating through a set of API endpoints, where several requests must be chained in a particular order to correctly fetch the relevant data. In addition, responses returned by a service can be paginated and thus necessitate several subsequent requests to accumulate all the data. Thus, constructing a state can quickly become laborious.

Once a state has been computed, it is necessary to detect changes with the previous one. However, off-the-shelf techniques can produce unexpected or irrelevant results as in the previous Facebook example in which photos with only a modified last update time should not be reported as different. Developing a generic differencing tool is a well-known complex problem, and can be NP-hard depending both on the change operations that are considered, and on the guarantees about the output size [6].

Our industrial partner, CProDirect, wishes to compete with traditional platforms by enabling fast integration of new service providers and events in its own platform [2]. To reduce time to market, we investigate the challenges of detecting changes in web service data. We focus on modern web services that follow the REST architectural style and exchange data with their consumers in JSON. We introduce a generative language-based approach, Polly, to simplify change detector construction.

Our contributions are the following:

- We introduce a new approach to change detector construction. Our approach relies on the use of a domain-specific language, Polly, for describing change detection strategies in JSON data fetched from REST web APIs.
- Our language provides declarative, simple yet highly-expressive constructs for describing how to construct a state from one or multiple API endpoints, how to identify changes in states, and how to produce a custom output.
- We have implemented a compiler that automatically produces an efficient JavaScript implementation which runs on top of a runtime system and hides low-level requirements such as HTTP authentication and pagination.

(a) Excerpt of a list of photos of a Facebook album.

(b) Excerpt of a list of tags of a Facebook photo.

Fig. 1: Excerpt of photos and tags from the Facebook service.

– We show the applicability of Polly by using it to automatically generate a number of change detectors for widely used web services such as Twitter, Facebook, and GitHub. We demonstrate that Polly's code is more concise that a manual implementation, and that it outperforms a state-of-the-art, off-the-shelf differencing technique.

The rest of this paper is organized as follows. Section 2 presents the range of issues that arise in detecting changes in web service data, as illustrated by a use case based on Facebook. Section 3 describes the Polly architecture and introduces a DSL for describing state construction, change detection and custom output construction. Section 4 demonstrates the efficiency and scalability of the Polly change detector. Section 5 discusses related work. Finally, Section 6 concludes and presents future work.

2 Challenges in service data change detection

To outline the multiple challenges involved when trying to detect changes in service data, we explain in details the scenario described in the introduction: *detecting new photos of a given Facebook album where Alice is tagged.*

In order to detect the new photos, one first needs to gather the complete list of photos of the Facebook album. This can be done by issuing a request on the `https://graph.facebook.com/v2.9/:albumId/photos` URL, where `:albumId` is the identifier of the photo album of interest. The Facebook service returns a response as a JSON document as illustrated in Fig. 1a. However, additional processing is needed to bridge the gap between the expected information and what is available in the returned document.

Firstly, the whole list of photos is not received at once, because the response is paginated (i.e. split in several lists of a fixed size). The `paging.next` attribute gives the URL to query to receive the next batch of photos. Additionally, the tags present on the photos are not part of this response. An additional request per

```
[
  {
    "created_time":
          ↪ "2016-05-20T12:26:57+0000",
    "updated_time":
          ↪ "2016-05-20T12:28:57+0000",
    "id": "1106290499393017",
    "data": [
      {
        "id": "10203528656797589",
        "name": "Bob",
        "created_time":
              ↪ "2016-05-20T12:39:01+0000",
        "x": 73.684210526316,
        "y": 74.865350089767
      }
    ]
  }
]
```


(a) Initial version.


```
[
  {
    "created_time":
          ↪ "2016-05-20T12:26:57+0000",
    "updated_time":
          ↪ "2016-05-20T12:29:57+0000",
    "id": "1106290499393017",
    "data": [
      {
        "id": "10203528656797589",
        "name": "Bob",
        "created_time":
              ↪ "2016-05-20T12:39:01+0000",
        "x": 76.684210526316,
        "y": 74.865350089767
      }
    ]
  },
  {
    "created_time":
          ↪ "2016-05-20T12:35:57+0000",
    "id": "2206280499393006",
    "data": [
    {
        "id": "20406528656797578",
        "name": "Alice",
        "created_time":
              ↪ "2016-05-20T12:45:57+0000",
        "x": 63.684210526316,
        "y": 62.865350089767
      }
    ]
  }
]
```


(b) Updated version.

Fig. 2: Initial and updated version.

photo is required to gather this information. This request can be made on the endpoint `https://graph.facebook.com/v2.9/:photoId/tags` where `:photoId` is the identifier of the photo of interest (received in response of the previous request). A request on the tags endpoint yields the result shown in Fig. 1b.

As we can see, this response is paginated as well. One can notice that the requests to gather the tags of each photo can be performed in an asynchronous manner, to improve performance. Finally, the tagged person names are available in these responses. To gather all the required information, the developer has then to manually construct a list that combines the photos and the tags data, as shown in Fig. 2a.

Performing a new polling operation using the same process would produce a new list of photos, as shown in Fig. 2b. By using an off-the-shelf differencing tool, the developer can compute the patch shown in Fig. 3. As it can be noticed, this patch contains two irrelevant changes: the `x` coordinate of the tag of the first photo and the last update time of the first photo. The only relevant change is the third one, where we can see a newly created photo containing a tag referring to user Alice. Therefore, the developer needs to post-process the patch produced by the differencing tool in order to construct the notification relevant to the scenario.

```
[
  {
    "op": "replace",
    "path": "/0/data/0/x",
    "value": 76.684210526316
  },
  {
    "op": "replace",
    "path": "/0/updated_time",
    "value": "2016-05-20T12:29:57+0000"
  },
  {
    "op": "add",
    "value":
    {
      "created_time": "2016-05-20T12:35:57+0000",
      "id": "2206280499393006",
      "data": [
      {
          "id": "0406528656797578",
          "name": "Alice",
          "created_time":
                ↪ "2016-05-20T12:45:57+0000",
          "x": 63.684210526316,
          "y": 62.865350089767
        }
      ]
    },
    "path": "/1"
  }
]
```


Fig. 3: JSON diff between the two versions of Figure 2.

In this example we clearly show that detecting changes in service data is a tedious operation. It requires navigating across several endpoints, possibly chaining response elements into query parameters, and handling the problem of pagination at each step. When the data is gathered, an off-the-shelf differencing tool may produce irrelevant changes thus requiring either post-processing of the output or developing an ad-hoc differencing algorithm.

3 Approach

As illustrated in Section 2, implementing custom change detectors of service data can be challenging for many developers. In this section, we introduce Polly, a declarative language-based approach that raises the level of abstraction by providing dedicated operators to express state construction, change detection, and output construction within a pipeline of operations. In the remainder of this section, we describe how our approach enables one to simply design efficient custom change detectors.

3.1 Overview of the Polly language

The Polly language is based on the YAML [18] syntax and is implemented as a Node.js module. Inspired by dataflow architectures, it is based on processing pipelines for defining custom change detectors. A pipeline is expressed as a series of transformation operations on successive sets of data, where data and operations on it are independent from each other. Each operation performs a specific task, and produces a JSON document that is passed as input for the following operation. Polly allows the user to specify how to compute a state by fetching a set of API resources, how to detect custom changes that are relevant to his requirements, and how to build a custom output to match the expected outcome. The provided language operators and constructs are described at greater length in the remainder of this section.

```
1  - operation: fetch
2    definition:
3      request:
4        url: https://graph.facebook.com
               ↪ /v2.9/:albumId/photos
5        params:
6          albumId: 465607303461343
7        query:
8          access_token: XXXX
9        headers:
10         Accept: application/json
11       template: ~.data
12       pagination:
13         next: ~.paging.next
14     # Or, instead of using template:
15     # output: &:$..data
```

(a) Polly specification for fetching a list of photos for a given album.

```
1  - operation: fetch
2    definition:
3      repeat:
4        forEach: _
5        placeholders:
6          photoId: ^.id
7      request:
8        url: https://graph.facebook.com
               ↪ /v2.9/:photoId/tags
9        query:
10         access_token: XXXX
11       headers:
12         Accept: application/json
13       pagination:
14         next: ~.paging.next
15       template:
16         photoId: ^.id
17         tags: ~.data
```

(b) Polly specification for fetching a list of tags for each album photo.

Fig. 4: A minimal example showcasing how to retrieve all photo tags of a Facebook album using Polly.

Language constructs. By design, each operation processes an input value (represented by the "`_`" symbol), and produces an output value (represented by the "`&`" symbol). These default values can be overridden using the *input* and *output* keywords at the operation level. Furthermore, Polly introduces two additional notations. The "`~`" symbol refers to the response body of a request (Fig. 4a, lines 11 and 13), while the "`%`" symbol refers to the response headers. The "`^`" symbol represents the loop iteration cursor (Fig. 4b, lines 6 and 16). This cursor represents the current element being iterated on. All five notations presented in this paragraph support the dot notation for accessing child properties. For example, `~.data` references the `data` attribute at the root of the response document.

Evaluating JSONPath expressions. Polly relies on the JSONPath [10] specification to describe the selection of a sub-document, as illustrated in line 15 of Fig. 4a. This enables users to easily extract the sub-documents of interest. Thus, a JSONPath expression[6] can be applied on any of the previous symbols, using the following notation: `[symbol]:[jsonpath_expr]`. For instance, the evaluation of the expression `&:$..id` is equivalent to evaluating `$..id` on the output document (`&`), thus producing all the `id` fields present in the output document.

3.2 State construction

The *fetch* operator enables the user to specify how to collect data from a set of API endpoints. These details are specified within the *request* block (Fig. 4a, line 3). Here, the user defines the resource URL using the *url* keyword (line

[6] The `$` symbol represents the root of the current document in JSONPath.

4). The URL can have parameter placeholders (prefixed by a colon), which are substituted with the matching key from the *params* block (line 5). Furthermore, the DSL offers the ability to specify query parameters (*query*, line 7) as well as HTTP headers (*headers*, line 9) as key-value pairs.

Templating. In the majority of use cases, the user only requires gathering a subset of the collected data. Furthermore, he might also need to include extra information along with the response. The *template* keyword allows specifying a transformation template. This can be expressed directly as an expression, or as a new set of keys where each corresponding value is an expression. For example, line 11 of Fig. 4a shows how to extract the `data` object from the API response (Fig. 1a, line 2). Another example occurs in line 15 of Fig. 4b where we fetch photo tags. Here, we define a new template containing the original photo ID and its tags. This transformation is necessary in order to manually include the photo ID (which is not part of the API response) in the final state.

Pagination. The *pagination* keyword enables the user to indicate how to fetch subsequent pages when the response is paginated (Fig. 4a, line 12). Information about pagination is typically present in an HTTP header or in the body of the response. For example, GitHub returns the full URL of the next page in the `Link` header, while Twitter provides just a cursor for the next page in the body of the response. Other APIs such as Stack Exchange require the user to manually specify the page number as a query parameter when requesting a resource, but do not provide any information about the current or next page number in the body of the response. Instead, they just indicate if there are subsequent pages using a boolean value in the body of the response. To support all these pagination methods, Polly enables the user to specify how to navigate to the following page using the *next* keyword (line 13). This keyword accepts either an expression containing the full URL of the next page, or key-value pairs specifying the name and value of the query parameter used for pagination (*queryParam*, defaults to the value `page` and auto-incremented by default). After collecting all subsequent pages, the results are flattened in a single array and returned as the output of the operation.

Parallel fetch. In the Facebook example presented in Section 2, the user has to first retrieve a list of photo IDs for a given album, then retrieve the tags for each photo. To enable this scenario, Polly provides the *repeat* keyword (Fig. 4b, line 3). This keyword allows specifying an iteration set from the output of the previous operation (*forEach*, line 4), and corresponding placeholder labels (*placeholders*, line 5). These placeholders are substituted in the URL by their value, thus executing a *request* for each constructed URL. In the Facebook example, this corresponds to fetching the tags for each album photo. By default, all requests are asynchronous and performed in parallel. The output of this operation contains a list of templated objects (line 15), where each object includes the current photo ID and the list of tags for a given photo (e.g. Fig. 1b).

```yaml
- operation: filterArray
  definition:
    input: _
    identifiers:
      - ^.photoId
    find:
      - addedItems
    output: &.addedItems
```

(a) Specification for detecting new photos.

```yaml
- operation: filterCustom
  definition:
    function: !!js/function >
      function (existing, input) {
        const result = [];
        input.forEach((item) => {
          const isTagged = item.tags.some((value) => {
            return value.name === 'Alice';
          });
          if (isTagged) { result.push(item.photoId); }
        });
        return { type: "addedTags", items: result };
      }
```

(b) Custom change detection specification.

Fig. 5: Detecting new photos where Alice is tagged using Polly.

3.3 Change detection

After computing the state in the previous step, the user can now proceed to specifying a change detection strategy. Our preliminary case studies showed that changes to a JSON document can occur on objects or arrays, and range from additions and suppressions, to value modifications and order changes. In light of these results, the Polly DSL provides several filtering operators for change detection: *filterObject*, *filterArray* and *filterCustom*. The *filterObject* (resp. *filterArray*) operator accepts an expression of object (resp. array) type as an input. The *filterCustom* operator enables the user to define custom filtering logic.

Change types. The *find* keyword enables defining a list of change types to detect in the input of the operation (Fig. 5a, line 6). The list of supported change types is presented in Table 1. For each change type listed in the *find* block, a matching object is included in the output of the operation, containing the corresponding data. For instance, listing *addedItems* and *removedItems* in the *find* block would produce as output an array of two objects, each having *addedItems* (resp. *removedItems*) as types, and each having a list of the items that have been detected as recently-added (resp. recently-removed).

Per-change type templating. Although the *template* keyword presented in Section 3.2 is also supported in this operation, one might need to specify different templates for different change types. To meet this requirement, Polly supports an additional keyword *templates* (mutually exclusive with *template*). This keyword allows specifying the change type (e.g. *addedItems*) as key, and the associated template as value.

Targeted monitoring. By default, all keys of the input document are watched for modifications, and any change would mark the document as modified. The optional keyword *watch* can be used to restrict the set of keys to watch for modifications. This enables the user to define what actually constitutes a relevant change. Note that for objects, a key is marked as modified (resp. unmodified) if

Table 1: List of supported change types.

	`filterObject`	`filterArray`
Change types	*addedKeys*	*addedItems*
	removedKeys	*removedItems*
	modifiedKeys	*modifiedItems*
	unmodifiedKeys	*unmodifiedItems*
		movedItems

the value corresponding to the key specified in the *watch* block is modified (resp. unmodified). For arrays, an item is marked as modified (resp. unmodified) if **any** (resp. **all**) of the values corresponding to the keys specified in the *watch* block are modified (resp. unmodified).

Custom item identification. Additionally, when dealing with array items, it is necessary to uniquely identify the items throughout subsequent polls. This allows us to know for example if a given item has been added or removed during the polling interval. However, not all APIs provide unique identifiers on all of their resources. Moreover, these identifiers can be present under different key labels. For this reason, we provide an additional keyword called *identifiers*, which allows the user to specify how to uniquely identify an item within a collection (line 4). This can be as simple as providing the path to the `id` field of an item, a list of fields (e.g. first and last names of a user), or a wildcard to hash the entire item and use it as its own identifier.

Custom filtering. When none of the previous operators are adequate, the *filterCustom* operator can be used to implement one's own custom filtering logic. Figure 5b shows an example of how to filter a list of photos by only selecting those where Alice is tagged. This operator provides a hook function with the previous and current states as parameters (line 4). The user can implement this hook in JavaScript, returning a custom output. In this example, the user iterates on the input array of photos (line 6) and checks whether if Alice is tagged on the current photo (lines 7-9), in which case he retrieves the photo ID (line 10). To avoid any security issues when running user-provided code, this function is executed within an isolated sandbox at runtime.

4 Evaluation

We evaluate our approach using six scenarios provided by our industrial partner CProDirect. We first compare the level of abstractions provided by Polly (in terms of verbosity) compared to its handwritten counterpart. We then assess the differencing time and the output size of our solution compared to a state-of-the-art differencing tool.

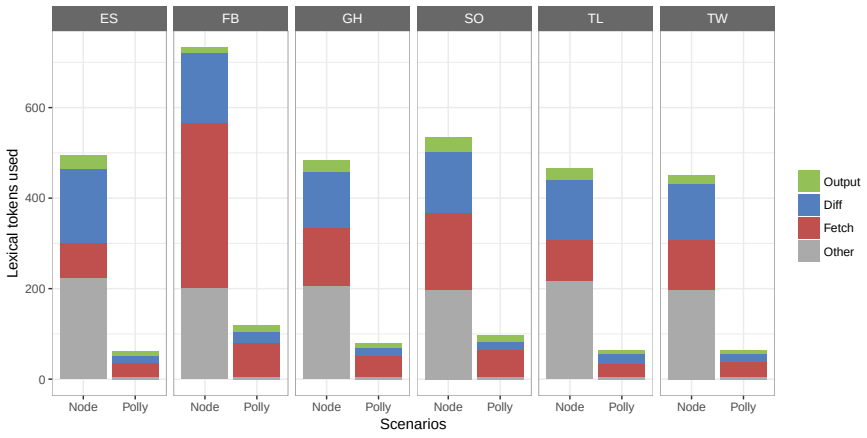

Fig. 6: Lexical tokens used to specify each scenario, using Node.js code *vs.* POLLY.

4.1 Scenarios

Our industrial partner CPRODIRECT has defined the six following scenarios to be used in our evaluation. They illustrate the diversity of possible use cases ranging from being notified about new objects to changes of attributes values or order in a ranking.

- **ElasticSearch (ES):** Developer Alice uses an instance of ElasticSearch as a search engine for her e-commerce platform, and wants to be notified when the top 5 best-selling products change in ranking order.
- **Facebook (FB):** Developer Alice wants to monitor a Facebook album where her friends Dan and Dave are participating. Alice would like to be notified only about pictures where Dan and Dave are tagged together.
- **GitHub (GH):** Developer Alice is interested in monitoring GitHub for new projects written in the Go language with over 2,000 stars.
- **Stack Overflow (SO):** Developer Alice wants to monitor StackOverflow for new JavaScript questions where there is an active bounty of over 100 reputation points.
- **Transport for London (TL):** Developer Alice wants to be notified whenever the status of the Victoria subway line changes (e.g. from healthy to faulty).
- **Twitter (TW):** Developer Alice wants to be notified whenever the official *Bordeaux* account has new followers on Twitter.

4.2 Language verbosity evaluation

All scenarios described in Section 4.1 have been implemented twice by the first author of the paper: once using the JavaScript language on top of the Node.js platform, and once using our domain-specific language POLLY. Note that the

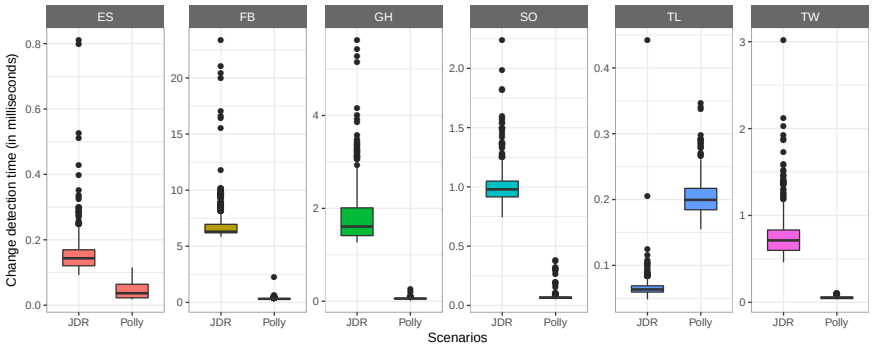

Fig. 7: Change detection time.

JavaScript version was implemented before any research work was done on Polly, in order to avoid any bias, and to serve as a reference point.

Figure 6 shows the number of lexical tokens used in the Node.js version versus the Polly version. One can notice that Polly results in a much smaller program, ranging from 5.5 to 8 times smaller. Furthermore, the figure shows the distribution of tokens across different categories (*fetch*, *diff* and *output*). Other tokens that are not directly related to these (such as module imports and configuration) are assigned to the *other* category. First, we notice that the Node.js implementation requires a lot more boilerplate code than Polly, with around 200 tokens in the *other* category, compared to 5 for Polly. Second, we notice that the *output* construction requires more or less the same number of tokens for both approaches, while it requires significantly less tokens for the *fetch* and *diff* categories using the Polly approach.

4.3 Diff performance experiment

Since one of the main benefits of using our approach is to be able to perform a custom differencing based upon domain knowledge of the data returned by the REST APIs, we wanted to evaluate in greater details the advantages of using such a strategy. We compare in this experiment the performance of Polly against a state-of-the-art generic differencing technique for JSON documents (JDR). We selected JDR as a candidate since prior benchmarks show it outperforms all other JavaScript differencing libraries [7].

Experimental setup. Since we are only focusing on the performance of the differencing and output construction stages for this benchmark, we can prefetch all required resources for better reproducibility. We thus proceeded to collect real data from the six service providers presented above. This is achieved by polling the services for the required resources over a period of 48 hours with an interval of 5 minutes, yielding 576 snapshots per service. We then serve this collected data through a mock server in the following experiments. All experiments were

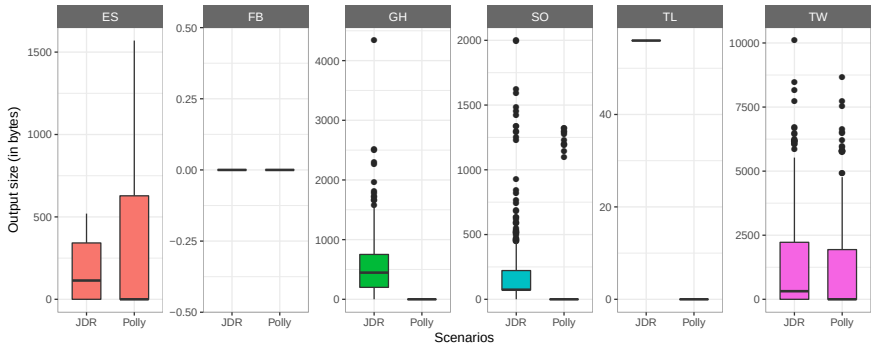

Fig. 8: Output sizes using JDR *vs.* Polly.

performed on a single machine powered by 8 GB RAM, and an Intel Core i7-6500U CPU @ 2.50GHz x 4.

Experimental protocol. We designed an experiment consisting in running each scenario 576 times (once for each snapshot) using JDR and Polly as change detection methods. At each step, we measure the differencing time as well as the output size. This process is repeated for 10 iterations for better precision. The results of this experiment are shown in Fig. 7 and Fig. 8. One can notice that the Polly approach produces lower differencing times and output sizes compared to the JDR approach, apart from the output size for the Facebook (FB) scenario, where the output size is equal to 0 for every polling step for both approaches. This is because no modifications occurred during the monitoring period. The difference in output sizes is explained by the fact that JDR produces a JSON Patch [5] (an intermediary document expressing a sequence of operations to apply to a JSON document in order to obtain the final outcome), whereas Polly directly produces the minimal set of required data as specified in the DSL, which generally tends to be much smaller in size.

Statistical testing. To have a finer-grained analysis of these results, we subject our results to a statistical testing. Our two null hypotheses are that H_0^1 output size is the same for Polly and JDR and H_0^2 differencing time is the same for Polly and JDR. Our two alternative hypotheses are H_a^1 output size is lesser for Polly than JDR and H_a^2 differencing time is lesser for Polly than JDR. To test these two hypotheses, we used a one-tailed paired Wilcoxon rank test, since it bears no assumptions on the underlying distribution of the differencing time or output size values. To assess the magnitude of the difference between differencing time and output size between the two approaches, we use Cohen's d and report its corresponding level on Cohen's standard scale. The results of this statistical testing are shown in Table 2.

One can notice that most tests are significant under the 0.05 threshold, meaning that Polly produces significantly smaller outputs in a significantly

Table 2: P-values of our statistical testing and size effect. Significant p-values (under the 0.05 threshold) are highlighted in bold.

	Detection time		Output size	
Scenario	P-value	Effect size (level)	P-value	Effect size (level)
ES	5.240281e-96	2.090851 (large)	4.447673e-42	0.685358 (medium)
FB	5.254766e-96	3.770390 (large)	1.000000e+00	NaN (NA)
GH	5.254964e-96	2.989436 (large)	1.362006e-84	1.186302 (large)
SO	5.254264e-96	6.150168 (large)	1.048446e-74	0.770466 (medium)
TL	1.000000e+00	-4.885277 (large)	6.361893e-99	88.626161 (large)
TW	5.254659e-96	2.846963 (large)	2.465265e-72	0.808057 (large)

reduced time compared to the JDR generic differencing approach. The only non-significant test is for the output size of the FB scenario. This is because in this scenario the output size is equal to 0 for every polling step for both approaches.

For the magnitude of the difference, the values range from medium to large, large being by far the most common value (9 times out of 11 values), followed by medium (2 times). This means that Polly results in a highly improved outcome in terms of output size and differencing time compared to the JDR approach.

5 Related work

Following the expansion of service-oriented computing, most service providers use the flexible REST architectural style to expose their data [8]. With web applications getting more and more complex, developers often need to navigate through multiple endpoints to retrieve the required resources. Existing efforts focused on a hypermedia-centric approach for describing REST services, using the Resource Linking Language (ReLL) and Petri Nets [1]. However, very few REST APIs provide hyperlinks along their responses in practice, making it harder for developers to gather all resources to compute a given state. To enable this case, our domain-specific language provides the necessary constructs to easily express sequential and parallel request chains of API endpoints, while also supporting pagination.

Due to the rapid growth of the number of web services in the recent decade, composition platforms are gaining more and more traction [12]. These platforms typically allow users to monitor third-party services in order to trigger a composition when a particular event occurs [16]. Thus, it is important to support a wide range of trigger events in order to meet the client's needs, scaling accordingly for all the services supported by the platform. Although previous works focused on providing a framework for automatic detection of relevant changes on websites [4], these do not directly address change detection in REST APIs data, nor do they allow clients to specify what constitutes a relevant change. In contrast, Polly offers a simple and concise language to rapidly specify custom change detectors, tailored to the user's expectations.

In today's fast-paced web, data is continuously churning to reflect the latest state. Change detection consists in computing a diff between two documents, and identifying any relevant changes. Several existing contributions focus on improving the differencing process. They represent documents as ordered or unordered labeled trees, and aim for optimizing the tree edit distance [3, 6, 19]. Nonetheless, the problem of finding a minimal patch is $O(n^3)$ to NP-hard for ordered trees (depending on the set of operations considered), and NP-hard for unordered trees [9, 14, 20]. This leads to the use of practical heuristics that rely on the syntactical properties of the documents in order to provide reasonably good results. As such, additional algorithms have been designed specifically for detecting changes in XML documents [17]. More recently, other algorithms have been designed for JSON documents, which are a combination of unordered and ordered labeled trees [7]. However, Polly relies on the client's business domain knowledge to finely tune the change detector. This improves the change detection process by enabling the selection of the most adequate strategy, thus discarding any irrelevant data.

With today's growing use of mobile devices, a particular focus is given to energy efficiency. Producing minimal diffs becomes particularly important when dealing with mobile clients, as it helps reducing the bandwidth usage [15]. Our approach addresses this concern by enabling the developer to specify the output resulting from the change detection process. This enables sending only the useful bits of information to the client, discarding all other irrelevant changes, thus reducing the payload size to the bare minimum.

6 Conclusion

Detecting custom changes in service data is a repetitive and tedious task. In this paper, we have presented Polly, a declarative domain-specific language for this task. Polly raises the level of abstraction by leveraging the business domain knowledge of users. It enables users to design custom change detectors by providing the necessary constructs to express state computation, change detection and output construction. We have used Polly to automatically generate custom change detectors for six use cases provided by our industrial partner CProDirect. Our evaluation shows that Polly outperforms a handwritten implementation in terms of code verbosity, and that Polly outperforms a state-of-the-art off-the-shelf differencing tool in terms of running time and output size. To showcase our solution, an online demonstration of Polly is freely available at the following address[7]. As future work, we plan on performing a large-scale developer study, where we assess the benefits of using Polly in terms of productivity, code quality and maintenance cost.

Acknowledgment. This work was partially supported by CProDirect and the French funding agency ANRT under contract CIFRE-2013/0891.

[7] `https://demo.pollyapp.ml`

References

1. Alarcon, R., Wilde, E., Bellido, J.: Hypermedia-driven RESTful service composition. ICSOC 6568 LNCS (2011)
2. Ben Hadj Yahia, E., Réveillère, L., Bromberg, Y.D., Chevalier, R., Cadot, A.: Medley: An Event-Driven Lightweight Platform for Service Composition. In: 16th International Conference on Web Engineering (2016)
3. Bille, P.: A survey on tree edit distance and related problems. Theoretical computer science 337(1) (2005)
4. Borgolte, K., Kruegel, C., Vigna, G.: Relevant change detection: a framework for the precise extraction of modified and novel web-based content as a filtering technique for analysis engines. In: 23rd International Conference on World Wide Web (2014)
5. Bryan, P., Nottingham, M.: Javascript object notation (json) patch. RFC 6902 (Proposed Standard) (2013)
6. Buttler, D.: A short survey of document structure similarity algorithms. In: International Conference on Internet Computing (2004)
7. Cao, H., Falleri, J.R., Blanc, X., Zhang, L.: JSON Patch for Turning a Pull REST API into a Push. In: International Conference on Service-Oriented Computing (2016)
8. Fielding, R.T.: Architectural styles and the design of network-based software architectures. Ph.D. thesis, University of California, Irvine (2000)
9. Higuchi, S., Kan, T., Yamamoto, Y., Hirata, K.: An A* algorithm for computing edit distance between rooted labeled unordered trees. In: JSAI International Symposium on Artificial Intelligence (2011)
10. JSONPath: `http://goessner.net/articles/JsonPath`, Accessed: 2017-06-02
11. Liu, L., Pu, C., Tang, W.: WebCQ-detecting and delivering information changes on the web. In: 9th International Conference on Information and Knowledge Management (2000)
12. Ovadia, S.: Automate the internet with "if this then that"(IFTTT). Behavioral & social sciences librarian 33(4) (2014)
13. Pandey, S., Dhamdhere, K., Olston, C.: WIC: A general-purpose algorithm for monitoring web information sources. In: 30th International Conference on Very Large Data Bases (2004)
14. Pawlik, M., Augsten, N.: RTED: a robust algorithm for the tree edit distance. VLDB Endowment 5(4) (2011)
15. Simon, J., Schmidt, P., Pammer, V.: An energy efficient implementation of differential synchronization on mobile devices. In: 11th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (2014)
16. Ur, B., Pak Yong Ho, M., Brawner, S., Lee, J., Mennicken, S., Picard, N., Schulze, D., Littman, M.L.: Trigger-action programming in the wild: An analysis of 200,000 IFTTT recipes. In: CHI Conference on Human Factors in Computing Systems (2016)
17. Wang, Y., DeWitt, D.J., Cai, J.Y.: X-Diff: An effective change detection algorithm for xml documents. In: 19th International Conference on Data Engineering (2003)
18. YAML: `http://www.yaml.org/spec/1.2/spec.html`, Accessed: 2017-06-02
19. Zhang, K., Shasha, D.: Simple Fast Algorithms for the Editing Distance Between Trees and Related Problems. SIAM J. Comput. 18(6) (Dec 1989)
20. Zhang, K., Statman, R., Shasha, D.: On the editing distance between unordered labeled trees. Information Processing Letters 42(3) (1992)